UNCLASSIFIED

AD NUMBER
ADB101693
LIMITATION CHANGES
TO: **Approved for public release; distribution is unlimited.**
FROM: **Distribution authorized to U.S. Gov't. agencies and their contractors; Critical Technology; FEB 1986. Other requests shall be referred to Rome Air Development Center, Griffiss AFB, NY 13441-5700. This document contains export-controlled technical data.**
AUTHORITY
RADC ltr 7 Jan 1988

THIS PAGE IS UNCLASSIFIED

AD-B101 693

(2)

RADC-TR-85-239
Final Technical Report
February 1986

VISIBLE LANGUAGES FOR PROGRAM VISUALIZATION

Sponsored by
Defense Advanced Research Projects Agency (DOD)
ARPA Order No. 4469

Dr. Ronald Baecker, Aaron Marcus, Michael Arent, Tracy Tims
and Allen McIntosh

DISTRIBUTION LIMITED TO U.S. GOVERNMENT AGENCIES AND THEIR CONTRACTORS; CRITICAL TECHNOLOGY; Feb 86. OTHER REQUESTS FOR THIS DOCUMENT SHALL BE REFERRED TO RADC (COEE), GRIFFISS AFB, NY 13441-5700.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

86 5 5 074

RADC-TR-85-239 has been reviewed and is approved for publication.

APPROVED: Andrew J Cruscicki

ANDREW J. CRUSCICKI
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Command and Control Division

FOR THE COMMANDER: Richard W. Pouliot

RICHARD W. POULIOT
Plans and Programs Division

VISIBLE LANGUAGES FOR PROGRAM VISUALIZATION

Contractors: Human Computing Resources Corp.
Aaron Marcus and Associates
Contract Number: F30602-83-C-0173
Effective Date of Contract: 20 October 1982
Contract Expiration Date: 30 September 1985
Short Title of Work: Program Visualization
Program Code Number: 4D30
Period of Work Covered: September 1982 - September 1985

Principal Investigator: Dr. Ronald Baecker
Phone Number: (416) 922-1937

RADC Project Engineer: Andrew Chruscicki
Phone Number: (315) 330-4063

Distribution limited to U.S. Government agencies and their contractors; critical technology, Feb 86. Other requests for this document shall be referred to RADC (COEE) Griffiss AFB NY 13441-5700.

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Andrew J. Chruscicki (COEE) Griffiss AFB NY 13441-5700, under Contract F30602-83-C-0173.

ADB101693

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT USGO agencies and their contractors; critical technology; Feb 86. Other requests RADC (COEE) Griffiss AFB NY 13441-5700.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-85-239
6a. NAME OF PERFORMING ORGANIZATION HCRC* Aaron Marcus Associates 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COEE)
6c. ADDRESS (City, State, and ZIP Code) 10 St. Mary Street, Toronto Ontario, Canada M4Y1P9 1196 Euclid Avenue, Berkeley CA 94706	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Defense Advanced Research Projects Agency 8b. OFFICE SYMBOL (If applicable) IPTO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-82-C-0173
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd Arlington VA 22209	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO.: 61101E PROJECT NO.: D469 TASK NO.: 01 WORK UNIT ACCESSION NO.: 02

11. TITLE (Include Security Classification)
VISIBLE LANGUAGES FOR PROGRAM VISUALIZATION

12. PERSONAL AUTHOR(S)
Dr. Ronald Baecker, Aaron Marcus, Michael Arent, Tracy Tims, Allen Mcintosh

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Sep 82 TO Sep 85	February 1986	

16. SUPPLEMENTARY NOTATION
*Human Computing Resources Corp.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
09	02	15	Program Visualization Software Maintenance; Pretty Printing. END

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes research to enhance the legibility and readability of C source text. Several practical results are presented. A graphic design manual documents a graphic design schema for the appearance of C source text. It is shown that many of the recommendations for C can be transferred to other languages, such as PASCAL and Ada. A prototype tool, called the SEE compiler, was developed for generating automatically improved appearance for most C programs. A prototype program book, that uniquely identifies the nature of information necessary to maintain a program, is also developed. (Keywords:

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Andrew J. Chruscicki	22b. TELEPHONE (Include Area Code): (315) 330-4065 22c. OFFICE SYMBOL: RADC (COEE)

Acknowledgments

Human Computing Resources Corporation (HCR) and Aaron Marcus and Associates (AM+A) carried out the program visualization research and prepared this final report.

Ron Baecker, Chairman of the Board of HCR, and Aaron Marcus, Principal of AM+A, were co-principal investigators, conceptualizing, structuring, and supervising the research. Michael Arent, Design Director of AM+A, played a key role throughout the research and in the development of the C language specifications and the preparation of the report. Baecker, Marcus and Arent did the conceptual work on prototype visualizations for the C language, and were assisted in their preparation by Bruce Browne, Designer at AM+A and John Longarini, programmer at HCR. Paul Breslin, Longarini, Allen McIntosh, Chris Sturgess, and Tracy Tims, programmers at HCR, wrote the software that enabled C programs to be compiled and displayed automatically in the manner recommended by the manual. David Slocombe, President, Soft Quad, Inc., Toronto, also contributed software development under contract to HCR. Baecker was the primary author of the bulk of the report, with significant collaboration from Marcus, Arent, Tims, and McIntosh.

The US Defense Department's Advanced Research Projects Agency, System Sciences Division, funded this project under ARPA Order 4469. We are indebted to Craig Fields, Clint Kelly, and Steve Squires of DARPA and Andrew Chruscicki of the Rome Air Development Center for their support.

We are particularly grateful to Michael, John, and Tracy for the extraordinarily long hours they spent in the final stages of completing this final report. We should also like to thank a number of individuals, Lu-anne Lee, Barb McLatchie, Janice Herrington, and Karyn Baecker, who assisted in the compilation and preparation of the report.

Preface

When you make a thing, a thing
    that is new, it is so complicated
    making it
that it is bound to be ugly.
But those that make it after you,
they don't have to worry
        about making it.
And they can make it pretty, and
    so everybody can like it
when the others
make it after you.

Picasso (as quoted by Gertrude Stein)

[From Victor Papanek (1982), *Design for the Real World*, London: Granada Publishing, p. 131.]

Page

Table of Contents

Page

List of Figures

Chapter 1

Introduction

The continuous and spectacular development of computer hardware that has occurred over the past four decades has finally been matched in recent years with corresponding advances in software engineering, that is, in the technology and processes of software development.

Typically, efforts have been made on a number of fronts. The most widespread development has been the concern with the logical structure and expressive style of programs. Out of this concern have emerged many of the modern software development techniques, including top-down design and stepwise refinement [Wirth, 1971], structured programming [Dahl, Dijkstra & Hoare, 1972], modularity [Parnas, 1972], and software tools [Kernighan & Plauger, 1976]. A second development has been the marked improvement in the clarity and expressive power of programming languages, as can be seen for example in Modula [Wirth, 1977]. Another kind of development has occurred in the organization and management of the team that produces the writing. This has given rise, for example, to the concepts of chief programmer teams [Baker, 1972] and structured walkthroughs [Yourdon, 1979].

The above advances have not been aided by progress in interactive computer graphics, but some other areas have benefited. It is now possible to construct interactive editors for various graphic notations that express algorithms and data structures, for example, Nassi-Schneiderman diagrams [Nassi & Schneiderman, 1973], Warnier-Orr diagrams [Higgins, 1979], contour diagrams [Organick & Thomas, 1974], and SADT diagrams [Ross, 1977]. (See [Martin & McClure, 1985] for a recent survey of these diagramming schemes and notations.) Even more significant is the increasing interest in enhancing the technology to support the writing and maintaining of good programs by providing, for example, integrated software development environments [Wasserman, 1981] such as INTERLISP [Teitelman, 1979] and high-performance personal workstations specialized to the task of program development [Gutz, Wasserman & Spier, 1981].

How have these developments improved the daily life of most programmers? Almost all have benefited from the use of modern programming languages. On the other hand, the impact of new software development methodologies, programmer team organizations, graphic diagramming notations, and sophisticated

programmer development environments has been limited for the most part to those working in research laboratories and in large corporate programming shops. Significant assistance has not yet been available to the lone programmer or small programming group who typically work in BASIC or C on systems of moderate complexity.

Section 1.1 Our Approach

We have taken a different approach in our recent work [Marcus & Baecker, 1982; Baecker & Marcus, 1983]. We focused on every programmer's vehicle of discourse: the program, expressed in some computer language and appearing in some form on some physical medium.

Since the advent of programming, the technologies of the video display terminal and the line printer have limited the presentation of a computer program's source code and comments to the use of a single type font, at a single point size, with fixed-width characters, and sometimes without even the use of upper and lower case. The technologies of high resolution bit-mapped displays, laser printers, and computer-driven phototypesetters, on the other hand, allow for the production of far richer representations, embodying multiple fonts, non-alphanumeric symbols, variable point sizes, variable character widths, proportional character spacing, variable word spacing and line spacing, gray scale tints, rules, and arbitrary spatial location and orientation of elements on a page. We therefore explore systematically in our work how these capabilities can be used to enhance the art of program presentation.

Our work thus encompasses the field of prettyprinting, an area in which others before us have worked with more limited graphics tools. The earliest work was done on LISP, so that program readers would not drown in a sea of parentheses. The problems of prettyprinting PASCAL have elicited a long correspondence in the ACM SIGPLAN notices [Hueras & Ledgard, 1977; Grogono, 1979; Gustafson, 1979; Leinbaugh, 1980]. A discussion of prettyprinting algorithms and their complexity has appeared [Oppen, 1980]. Other authors [Rose & Welsh, 1977; Rubin, 1983] demonstrated methods of extending the syntactic descriptions of programming languages to include their formatting conventions. One paper [Miara, Musselman, Navarro & Schneiderman, 1983] includes a review of a number of human factors experiments concerning the effect of program indentation on program comprehensibility. Unfortunately, these experiments have generally failed to provide experimental confirmation of what every programmer knows: a program's appearance dramatically effects its comprehensibility and useability.

Our work however goes significantly beyond suggesting recommended conventions for appearance that enhance the prettyprinting of program code. We have also developed a flexible tool with

which future programmers and human factors specialists may tune and improve these conventions, thus paving the way for successful standards. In addition, we have considered the entire context in which code is presented, a context which includes the supporting texts and notations that make a program a living piece of written communication.

Section 1.2 Programs as Publications

Programs are publications, a form of literature. Just as English prose can range in scope from a note scribbled on a pad to a historical treatise appearing in multiple volumes and representing a lifetime of work, so do we find a variety of programs ranging from a two line *shell* script created whenever needed to an edition of the collected program works of a laboratory, as is the case, for example, with the UNIX (tm) operating system. (See [Lions, 1977] for an early example of this idea applied to the UNIX kernel.) The line printer listing, which represents the output of conventional program publishing technology, is woefully inadequate for documenting an encyclopedic collection of code such as the UNIX system, or even for such lesser program treatises as compilers, graphics subroutine packages, and data base management systems.

What we have done, therefore, is to apply the tools of modern computer graphics technology and the visible language skills of graphic design, guided by the metaphors and precedents of literature, printing, and publishing, to suggest and demonstrate in prototype form that enduring programs should and can be made more accessible and more useable.

We divide the content of a program into three kinds of text: primary, secondary, and tertiary. Primary text includes what typically appears in a program listing: the program code and comments. Secondary text includes various metadata describing the context in which the program is used and various short commentaries (often mechanically produced) pointing out salient features of the program. Tertiary text includes the various longer descriptions and explanations of the program that typically are called documentation.

(tm) UNIX is a trademark of AT&T Bell Laboratories.

Section 1.3 The Goal of Our Research

Our goal has been to take a fresh approach to the presentation of source text, and thereby to make it:

— more legible

— more readable

— more intelligible

— more vivid

— more appealing

— more memorable

— more useful

— more maintainable.

Section 1.4 Methodology of Our Research

Our research has proceeded as follows:

We first developed a graphic design taxonomy for computer-based documents and publications. This was intended to be a checklist for approaches to enhancing source code presentation [Gerstner, 1978; Ruder, 1973; Chaparos, 1981].

We simultaneously developed a taxonomy of C constructs, a systematic enumeration and classification of aspects of the language [AT&T, 1985; Kernighan & Ritchie, 1978; Harbison & Steele, 1984]. This was intended to be a companion checklist for insuring completeness in the representation of C source text. We subsequently reworked our taxonomy slightly to make it maximally consistent with the presentation in [Harbison & Steele, 1984]. We chose to work with C for a number of reasons: its commercial importance, its illegibility, and its unreadability.

Next, we collected and systematized typical mappings from C constructs to typographic constructs, examples abstracted from real C programs prepared by typical experienced C programmers. Because these examples often embody real design insights from non-designers, we call them "folk designs".

Then, we developed a systematic approach to the design of mappings from C constructs to typographic constructs, an approach that forms the basis for detailed visual research into effective presentations of C source code. We shall describe the approach in detail in this report and illustrate it via an application to a concrete example.

To test our systematic approach to the design of program presentation, we constructed SEE, a visual C compiler, a program that maps an arbitrary C program into an effective typeset representation of that program. A description of the implementation appears in Volume 6 of the report. We have produced numerous examples using this automated tool, which has in turn enabled us to improve the graphic design of program appearance. Some of the examples are collected in Volume 3 of the report. The final specifications were then embodied in a graphic design manual for the appearance of C programs. This manual is Volume 2 of the report.

Finally, we shifted our viewpoint away from the details of code

appearance and considered the larger issue of the function, structure, contents, and form of the *program book*, the embodiment of the concept of the program as a publication. Although we did not fully automate its production, we developed and have included as Volume 5 of the report a mock-up of a prototype of a program book. For comparison purposes, we have included as Volume 4 "the same" listings and documentation in the form in which programmers would currently receive it.

Section 1.5 The Final Report and the Deliverables

Volume 1: Theory, Results, and Conclusions

This volume presents the theory, summarizes the results, and suggests the conclusions that may be derived from the overall work.

Volume 2: A Graphic Design Manual for C

Volume 2 summarizes our systematic approach to the design of program presentation from a graphic design perspective. It is therefore a graphic design manual for the appearance of C programs and C program books.

Volume 3: Graphic Design Variations of C Program Appearance

Volume 3 presents selected examples of C program visualization that can be realized with the SEE program visualizer and that present significant variations of the recommended conventions.

Volume 4: Traditional Listings and Documentation for the Eliza Program

Volume 4 presents the listings and documentation for a program in its typical form of appearance. The program shown is Joseph Weizenbaum's famous Eliza program [Weizenbaum, 1966]. Henry Spencer of the Department of Zoology of the University of Toronto has implemented this new version.

Volume 5: A Prototype Program Book of the Eliza Program

Volume 5 illustrates the concept of the program as a publication. A mock-up of a prototype program book of the Eliza program appears. Included in the mock-up is the primary source text, the code and comments, which were automatically typeset by the SEE program visualizer.

Volume 6: A Program Visualization Implementation

Volume 6 describes the implementations of SEE and of the UNIX TROFF [Kernighan, 1982] typesetting macro packages used to format program visualization text and programs.

Deliverables

These six volumes comprise the Final Report and the Graphic Design Manual to be delivered to DARPA as per the Contract Data Requirements List of Contract Number F30602-82-C-0173. In particular, referring back to the Statement of Work, Section 4.2, the "typeset examples" of Section 4.2.1 are included in our Volumes 1 through 3 and 5; the "program" of Section 4.2.2 is described in our Volumes 1 and 6, the "Graphic Design Manual" of Section 4.2.3 is our Volume 2; and, the "report" and "image sequences" of Section 4.2.4 are included in our Volumes 2 through 5.

A Program Visualization video tape is being prepared which illustrates the objectives, goals, method, results, and significance of our work in a more informal manner. A magnetic tape containing the implemented program is available where appropriate.

Finally, we note that the typeset examples in Volumes 1, 3, and 5 wre prepared "almost totally automatically" by SEE. Electronic or manual fix-ups were used to fix three bad line breaks in Volume 5, to add some white space in two recurring kinds of locations in Volumes 1 and 5, to fix roughly six bad page breaks in Volumes 1 and 5, to add letratone, an occasional bracket, and the pointing fingers that appear in Volumes 1, 3, and 5, and to add the footnotes shown in Figure 50 of Volume 3. For comparison purposes, fingers have only been used in the example in Volume 1, the first five figures in Volume 3, and one file of Eliza in Volume 5.

Chapter 2

An Example of the Design of Program Appearance

Our example consists of a slightly updated version of a desk calculator program that appears in a standard book on C [Kernighan & Ritchie, 1978].

The program is shown as Figure 1 on pages 16 through 18 as it is output on a typical dot matrix line printer, a device similar to that used by tens of thousands of programmers of microcomputers and minicomputers. Even the lightness of the type, caused by a worn out ribbon, reflects an unfortunate aspect of the way most line printers are used. This of course impedes legibility and readability.

The program is shown again as Figure 2 on pages 20 through 22. This time it has been output on a modern laser printer. It appears in exactly the same format as does Figure 1, and again uses fixed width type in a single font at a single point size. Legibility and readability are somewhat enhanced.

Figure 3 on pages 24 through 27 shows the output from the current version of the SEE processor to the same laser printer with an appropriate set of fonts. The C program was not modified at all for input to SEE; exactly the same text was input to the listing program that produced Figures 1 and 2. The SEE output was massaged only in the introduction of some white space to improve the way in which the program is paginated, since white space introduction and pagination are not yet handled automatically by SEE. The subtitles below refer to categories of program visualization improvements discussed later in this volume; the numbers in the margin of Figure 3 refer to various items in the following commentary:

The Presentation of Program Metadata

1. The program is presented on a standard 8½x11 inches page that is separated into four regions, a header, a footnote area, a code column, and a marginalia comment column.

2. The header contains key document metadata describing the context of the source code that appears on the page, including the location of the file from which the listing was made and the page number within the listing.

The Spatial Composition of Comments

3. Comments that are *external* to function definitions are displayed in a small-sized serif font inside an outline box. There is ample margin allowance around the text to ensure optimum legibility and readability.

4. Comments that are *internal* to function definitions are displayed in a small-sized serif font appropriately indented and marked by a left vertical bracket.

5. Comments that are located on the same lines as source code, which we call *marginalia* comments, are displayed in a small-sized serif font in the marginalia column. These items are intended to be short single line phrases.

The Typography of Program Punctuation

6. In this example the ";" appears in 10 point regular Helvetica type, and thus uses the same typographic parameters as does much of the program code. The ":", on the other hand, has been set in bold type, and the "," has been enlarged to 14 point. These distinctions highlight the difficulties in achieving legible punctuation with currently available typefaces. The bold is often slightly too heavy; the regular weight is sometimes too easily overlooked if the original has been poorly displayed with badly adjusted equipment or if it has been degraded through photocopying. In addition, idiosyncratic size changes for particular characters in particular fonts are often desirable.

7. Symbols such as the "++" and the "−−" have been kerned, that is, the letter spacing of individual characters overlaps to make them more legible and readable.

8. Symbol substitutions have not been introduced for symbols that clearly need improved appearance, e.g., the ">=", and "==". Whether or not these substitutions are invoked should be determined by a flag under control of the user. Legibility criteria would suggest innovation; however, reader familiarity and direct semantic reference to two input keyboard strokes would suggest the conventional alternative that we currently recommend. For an example of this, see Volume 3, Figure 20, page 28.

Typographic Encodings of Token Attributes

9. Most tokens are shown in a regular sans-serif font; reserved words are shown in italic sans-serif type. Bold sans-serif is used to highlight global (*extern*) variables (see 22).

10. String constants are shown in a small-sized serif font.

The presentation of Preprocessor Commands

11. The "#" signifying a preprocessor command is exdented to enhance its distinguishability from ordinary C source text.

12. Macros and their values are presented at appropriate horizontal tab positions.

The Presentation of Declarations

13. Identifiers being declared are aligned to a single implied vertical line located at an appropriate horizontal tab position.

The Visual Parsing of Expressions

14. Parentheses and brackets are emboldened to call attention to grouped items. Nested parentheses are varied in size to aid the parsing of the expression.

15. The word spacing between operators within an expression is varied to aid the visual parsing of the expression. Operands are displayed closer to operators of high precedence than to operators of low precedence.

The Visual Parsing of Statements

16. Systematic indentation and placement of key words is employed.

17. Since curly braces are redundant with systematic indentation, they are removed in this example. Whether this happens or not is determined by a flag under control of the user.

18. "Unusual" control flow is marked with pointing figures located in the margin.

The Presentation of Function Definitions

19. The introductory text of a function definition, that is, the function name, is shown in bold sans-serif type.

20. A heavy rule appears under the introductory text of a function definition.

21. A light rule appears under the declaration of the formal parameters.

The Presentation of Program Structure

22. The global variable in C is a fundamental mechanism through which functions can communicate indirectly, and as such also represents a major potential source of programming errors. We therefore call attention to most uses of globals (but not manifest constants) by highlighting them in bold face.

23. Cross-references relating identifiers used in one file to the location of their definitions in another file could be included as footnotes to the source text. For an example of this, see Volume 3, Figure 50, page 65.

Figure 1: A listing of a simple desk calculator program produced on a dot matrix line printer

(See next 3 pages.)

```
/*
This reverse Polish desk calculator adds, subtracts, multiplies and
divides floating point numbers.  It also allows the commands '=' to
print the value of the top of the stack and 'c' to clear the stack.
*/
#include <stdio.h>
#define MAXOP 20        /* max size of operand, operator */
#define NUMBER '0'      /* signal that number found */
#define TOOBIG '9'      /* signal that string is too big */


/*                      Control Module                          */

calc()
{
        int type;                                   /* operation type */
        char s[MAXOP];                              /* buffer containing operator */
        double  op2,                                /* temporary variable */
                atof(),                             /* converts strings to floating point */
                pop(),                              /* pops the stack */
                push();                             /* pushes the stack */

        /* loop while we can get an operation string and type */

        while ((type = getop(s, MAXOP)) != EOF)
                switch (type){
                case NUMBER:
                        push(atof(s));
                        break;
                case '+':
                        push(pop() + pop());
                        break;
                case '*':
                        push(pop() * pop());
                        break;
                case '-':
                        op2 = pop();
                        push(pop() - op2);
                        break;
                case '/':
                        op2 = pop();
                        if (op2 != 0.0)
                                push (pop() / op2);
                        else
                                printf("zero divisor popped\n");
                        break;
                case '=':
                        printf("\t%f\n", push(pop()));
                        break;
                case 'c':
                        clear();
                        break;
                case TOOBIG:
                        printf("%.20s ... is too long\n", s);
                        break;
                default:
                        printf("unknown command %c\n", type);
                        break;
                }
}


/*                  Stack Management Module                     */

#define MAXVAL 100      /* maximum depth of val stack */

int sp = 0;             /* stack pointer */
double val[MAXVAL];     /* value stack */

double push(f)          /* push f onto value stack */
double f;
{
        if (sp < MAXVAL)
                return (val[sp++] = f);
        else {
             printf("error; stack full\n");
             clear();
```

```
		return(0);
	}
}

double pop()		/* pop top value from stack */
{
	if (sp > 0)
		return(val[--sp]);
	else {
		printf("error: stack empty\n");
		clear();
		return(0);
	}
}

clear()			/* clear stack */
{
	sp = 0;
}


/*			Input Module				*/

getop(s, lim)		/* get next operator or operand */
char s[];		/* operator buffer */
int lim;		/* size of input buffer */
{
	int i, c;

	/* skip blanks, tabs and newlines */

	while ((c = getch()) == ' ' || c == '\t' || c == '\n')
		;

	/* return if not a number */

	if (c != '.' && (c < '0' || c > '9'))
		return(c);
	s[0] = c;

	/* get rest of number */

	for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
		if (i < lim)
			s[i] = c;
	if (c == '.') {		/* collect fraction */
		if (i < lim)
			s[i] = c;
		for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
			if (i < lim)
				s[i] = c;
	}
	if (i < lim) {		/* number is ok */
		ungetch(c);
		s[i] = '\0';
		return(NUMBER);
	} else {		/* it's too big; skip rest of line */
		while (c != '\n' && c != EOF)
			c = getchar();
		s[lim - 1] = '\0';
		return(TOOBIG);
	}
}


#define BUFSIZE 100

char buf[BUFSIZE];	/* buffer for ungetch */
int bufp = 0;		/* next free position in buf */

getch()			/* get a (possibly pushed back) character */
{
	return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c)		/* push character back on input */
int c;
{
```

```
        if (bufp > BUFSIZE)
                printf("ungetch: too many characters\n");
        else
                buf[bufp++] = c;
}
```

Figure 2: A listing of the desk calculator program produced on a laser printer

(See next 3 pages.)

```
/*
This reverse Polish desk calculator adds, subtracts, multiplies and
divides floating point numbers.  It also allows the commands '=' to
print the value of the top of the stack and 'c' to clear the stack.
*/
#include <stdio.h>
#define MAXOP 20        /* max size of operand, operator */
#define NUMBER '0'      /* signal that number found */
#define TOOBIG '9'      /* signal that string is too big */


/*                      Control Module                          */

calc()
{
        int type;                               /* operation type */
        char s[MAXOP];                          /* buffer containing operator */
        double  op2,                            /* temporary variable */
                atof(),                         /* converts strings to floating point
                pop(),                          /* pops the stack */
                push();                         /* pushes the stack */

        /* loop while we can get an operation string and type */

        while ((type = getop(s, MAXOP)) != EOF)
                switch (type){
                case NUMBER:
                        push(atof(s));
                        break;
                case '+':
                        push(pop() + pop());
                        break;
                case '*':
                        push(pop() * pop());
                        break;
                case '-':
                        op2 = pop();
                        push(pop() - op2);
                        break;
                case '/':
                        op2 = pop();
                        if (op2 != 0.0)
                                push (pop() / op2);
                        else
                                printf("zero divisor popped\n");
                        break;
                case '=':
                        printf("\t%f\n", push(pop()));
                        break;
                case 'c':
                        clear();
                        break;
                case TOOBIG:
                        printf("%.20s ... is too long\n", s);
                        break;
                default:
                        printf("unknown command %c\n", type);
                        break;
```

```
        if (c == '.') {            /* collect fraction */
                if (i < lim)
                        s[i] = c;
                for (i++; (c = getchar()) >= '0' && c <= '9'; i++)
                        if (i < lim)
                                s[i] = c;
        }
        if (i < lim) {             /* number is ok */
                ungetch(c);
                s[i] = '\0';
                return(NUMBER);
        } else {                   /* it's too big; skip rest of line */
                while (c != '\n' && c != EOF)
                        c = getchar();
                s[lim - 1] = '\0';
                return(TOOBIG);
        }
}

#define BUFSIZE 100

char buf[BUFSIZE];          /* buffer for ungetch */
int bufp = 0;               /* next free position in buf */

getch()                     /* get a (possibly pushed back) character */
{
        return((bufp > 0) ? buf[--bufp] : getchar());
}

ungetch(c)                  /* push character back on input */
int c;
{
        if (bufp > BUFSIZE)
                printf("ungetch: too many characters\n");
        else
                buf[bufp++] = c;
}
```

```
/*                    Stack Management Module                          */

#define MAXVAL 100          /* maximum depth of val stack */

int sp = 0;                 /* stack pointer */
double val[MAXVAL];         /* value stack */

double push(f)              /* push f onto value stack */
double f;
{
        if (sp < MAXVAL)
                return (val[sp++] = f);
        else {
                printf("error: stack full\n");
                clear();
                return(0);
        }
}

double pop()                /* pop top value from stack */
{
        if (sp > 0)
                return(val[--sp]);
        else {
                printf("error: stack empty\n");
                clear();
                return(0);
        }
}

clear()                     /* clear stack */
{
        sp = 0;
}


/*                       Input Module                                  */

getop(s, lim)               /* get next operator or operand */
char s[];                   /* operator buffer */
int lim;                    /* size of input buffer */
{
        int i, c;

        /* skip blanks, tabs and newlines */

        while ((c = getch()) == ' ' || c == '\t' || c == '\n')
                ;

        /* return if not a number */

        if (c != '.' && (c < '0' || c > '9'))
                return(c);
        s[0] = c;

        /* get rest of number */

        for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
                if (i < lim)
                        s[i] = c;
```

Figure 3: The desk calculator program produced on a laser printer using the SEE program visualizer

(See next 4 pages.)

1

Chapter 1 calc1.c

This reverse Polish desk calculator adds, subtracts, multiplies and divides floating point numbers. It also allows the commands '=' to print the value of the top of the stack and 'c' to clear the stack. 3

	# include	<stdio.h>		
Max size of operand, operator	# define	MAXOP	20	
Signal that number found	# define	NUMBER	'0'	5
Signal that string is too big	# define	TOOBIG	'9'	11

Control Module

calc()

Operation type	*int*	type;	
Buffer containing operator	*char*	s[MAXOP];	
Temporary variable	*double*	op2,	
Converts strings to floating point		atof(),	
Pops the stack		pop(),	
Pushes the stack		push();	

Loop while we can get an operation string and type 4

```
while ((type = getop(s, MAXOP)) != EOF)
      switch (type)
      case NUMBER:
            push(atof(s));
            break ;
      case '+':                                  6
            push(pop() + pop());
            break ;
      case '*':
            push(pop() * pop());                 14
            break ;
      case '-':
            op2 = pop();
            push(pop() - op2);
            break ;
      case '/':
            op2 = pop();
            if (op2 != 0.0)
                  push(pop() / op2);
            else
                  printf("zero divisor popped\n");
            break ;
```

```
        case '=':
            printf("\t%f\n", push(pop()));
            break ;
        case 'c':
            clear();
            break ;
        case TOOBIG:
            printf("%.20s ... is too long\n", s);
            break ;
        default :
            printf("unknown command %c\n", type);
            break ;
```

Stack Management Module

Maximum depth of val stack	#define	MAXVAL	100	12
Stack pointer	int		sp = 0;	13
Value stack	double		val[MAXVAL];	

Push f onto value stack

```
double
push(f)
    double                      f;

    if (sp < MAXVAL)                                  9
        return (val[sp++] = f);
    else
        printf("error: stack full\n");                10
        clear();
        return (0);
```

Pop top value from stack

```
double
pop()
    if (sp > 0)
        return (val[--sp]);                           22
    else
        printf("error: stack empty\n");
        clear();
        return (0);
```

Clear stack

```
clear()
    sp = 0;
```

Input Module

getop(s, lim) 19

Get next operator or operand 20

Operator buffer / Size of input buffer

```
char    s[];
int     lim;
```

21

```
int     i,
        c;
```

Skip blanks, tabs and newlines

```
while ((c = getch()) == ' ' || c == '\t' || c == '\n');
```

Return if not a number

```
if (c != '.' && (c < '0' || c > '9'))
        return (c);                                          18
s[0] = c;
```

Get rest of number

```
for (i = 1;  (c = getchar()) >= '0' && c <= '9';  i++)       16
        if (i < lim)                                         17
                s[i] = c;
```

Collect fraction

```
if (c == '.')
        if (i < lim)
                s[i] = c;
        for (i++;  (c = getchar()) >= '0' && c <= '9';  i++)  7
                if (i < lim)
                        s[i] = c;
```

Number is ok

```
if (i < lim)
        ungetch(c);
        s[i] = '\000';
        return (NUMBER);
```

It's too big; skip rest of line

```
else
        while (c != '\n' && c != EOF)                        15
                c = getchar();
        s[lim - 1] = '\000';
        return (TOOBIG);
```

Buffer for ungetch / Next free position in buf

```
#define         BUFSIZE         100
char                            buf[BUFSIZE];
int                             bufp = 0;
```

getch()

Get a (possibly pushed back) character

```
return ((bufp > 0)  ?  buf[--bufp]  :  getchar());
```

Push character back on input

ungetch(c)

```
int                                     c;

if (bufp > BUFSIZE)
        printf("ungetch: too many characters\n");
else
        buf[bufp++] = c;
```

Chapter 3

C Program Books

A program book would typically be composed of primary, secondary, and tertiary texts structured into five parts (see Figure 4):

— The book begins with secondary text known as the "front matter". This may include a cover page, title page, copyright page, abstract, authors and personalities page, and program history page.

— Chapter 1 is the tertiary text that comprises the user documentation: the command summary and manual page, the tutorial guide, and the reference manual.

— Chapters 2 through n+1 constitute the primary text, the program code and comments. Each file of the n files in the program appears in a separate chapter. Each program page has various metadata and commentaries included in its header and footer.

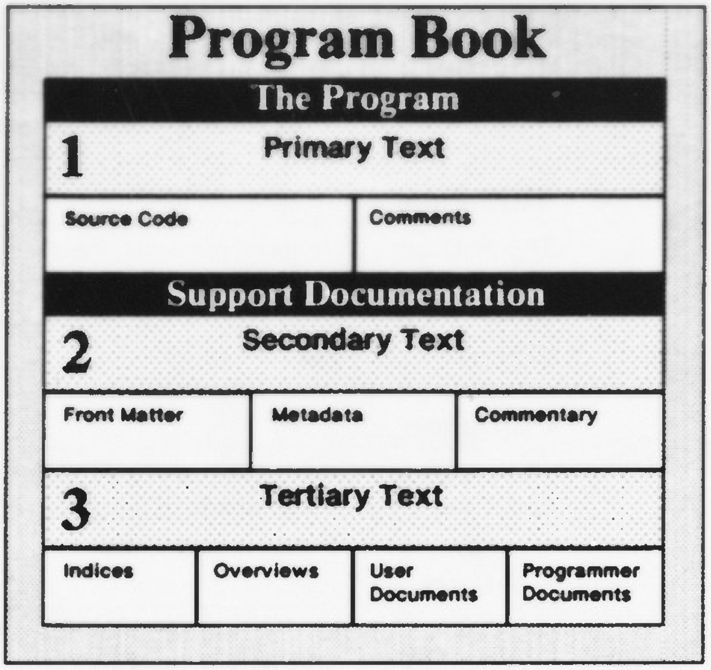
— Chapter n+2 contains more secondary text, various indices and overviews. These may include program metrics, program signatures and condensations, a cross reference index, a key word in context index, a call hierarchy, and various other diagrams.

— Chapter n+3 includes the remaining part of the tertiary text, the programmer documentation: the installation guide and README file, the "make" file, and the maintenance guide.

Whereas any listing or representation of the program or of a piece of it will contain primary text, some or most of these secondary texts can and will be omitted in a "quick and dirty" look at a program that is likely to be changed almost immediately, as is the case when one is creating or debugging code.

The tertiary text is the source of still additional information about the program, how it was built, and how it is to be used. Even more so than in the case of secondary text, the investment in the production of tertiary text is most easily justified if the program has considerable readership and longevity.

Figure 4: The structure of a program book



Section 3.1

Secondary Text: Front Matter

Cover Page

A program published in book form may need a cover page identifying the book and depicting it with an attractive illustration.

Title Page

The program's title page presents the most important metadata, such as the program's title, author, company and address of the author, version, date, publishing source, and level of confidentiality.

Colophon

The program's colophon presents production information, details about the typesetting, printing, and distribution of the document.

Abstract

An abstract of the program summarizes what it does, how it accomplishes it, and why it does it.

Program History

A design history presents the history of the system from conception to implementation through recent modification. As program genealogy, it may also be invaluable in understanding apparently nonsensical constructs and bizarre artifacts.

Authors and Personalities

This page lists the authors and other important personalities (e.g., augmenters and maintainers) associated with the program, gives their postal and network addresses, their phone numbers, and potentially also their photographs [Pike, 1985].

Table of Contents

The table of contents enumerates the major parts of the program. In the case of a program operating under the UNIX operating system, for example, it would probably list the directories and files and possibly also the defined functions.

Section 3.2

Tertiary Text: User Documentation

Command Summary and Manual Page

A summary of commands is essential for every user of any system. In the UNIX world, this command summary is often included in the manual page, or "man page". By convention, one such page is written to correspond to each UNIX utility or command installed on the system.

Tutorial Guide

A tutorial guide presents a step-by-step introduction to the usage of the major features of the system.

Reference Manual

A reference manual is a comprehensive information source on all features of the system.

Section 3.3

Primary Text: The Program

The primary text is the program itself. Its appearance is the topic of the next Chapter of this report. Each file of the program is represented by a number of program pages. These pages each include:

Program Code

The "program books" of today, known as listings, often contain only code.

Program Comments

Comments appear in various forms and locations on the page, as discussed in Chapter 4.2 of this volume.

Section 3.4

Secondary Text: Metadata and Commentaries

Also located on the program pages are two kinds of secondary text, selected metadata and program cross-reference information.

Program Page Headers

Program page headers include selected metadata under the control of the user requesting the listing.

Program Page Footnotes

Program page footnotes should include cross-references to the definitions of identifiers declared "externally" to that particular file.

Section 3.5

Tertiary Text: Indices and Overviews

Program Metrics

A list of *metrics* [Gilb, 1976; Perlis, Sayward & Shaw, 1981] would include numerical tables and charts encapsulating significant properties or qualities of the program. Software engineers and human factors specialists must determine their proper content.

Program Signatures and Condensations

Program *signatures* and program *condensations* are visual representations of the code that compress the text into small diagrams or symbols. These allow a viewer to quickly scan many pages of a program.

Cross Reference Index

Cross reference listings detail where every identifier is declared and all instances of its use.

Key Words in Context Index

Key word in context listings show all program phrases alphabetically in the context of their surrounding text.

Call Hierarchy

A *call hierarchy* diagram shows the nesting of function calls.

Other Diagrams

Various other diagrammatic representations [Martin & McClure, 1985] that portray the structure of the program should also be included.

Section 3.6

Tertiary Text: Programmer Documentation

The Installation Guide and README File

An installation guide contains instructions on how to install a system. In a UNIX distribution, it is typically part of a "README" file. In the UNIX world, a README file is by convention included on any tape containing a software distribution. This file is the first read by the programmer upon receipt of the system, and thus should be a guidebook to what is in the distribution.

The Make File

In the UNIX world, the "make" file is used by the UNIX "make" program to facilitate system recompilation and regeneration.

Maintenance Guide

The maintenance guide contains instructions on how to maintain the system. It is thus an additional commentary on the program.

Chapter 4

Graphic Design of C Source Code and Comments

Our goal in the research was to apply the full palette of graphic design techniques to reveal and express the meaning of C programs. We worked on ten specific problems and explored various methods for displaying the following:

The Presentation of Program Metadata

Enhancing the display of a program in relationship to the relevant data describing the context in which the program was created, is maintained, and will be used.

The Spatial Composition of Comments

Presenting program comments clearly in relationship to program code.

The Typography of Program Punctuation

Enhancing the visual effectiveness of C punctuation marks (separators, containment symbols, and operators).

Typographic Encodings of Token Attributes

Mapping C tokens (identifiers, reserved words, and constants) into effective typographic representations.

The Presentation of Preprocessor Commands

Presenting C preprocessor commands in a more effective manner.

The Presentation of Declarations

Enhancing the structure of the declarations of C identifiers.

The Visual Parsing of Expressions

Using typographic attributes to enhance the ability of a human reader to identify and understand complex program expressions.

The Visual Parsing of Statements

Using typographic attributes to enhance the ability of the reader to identify and understand complex program statements.

The Presentation of Function Definitions

Clarifying the structure of the definitions of C functions.

The Presentation of Program Structure

Enhancing the structure of a program in terms of its constituent parts, for example, its constituent files, declarations, and function definitions.

Section 4.1

The Presentation of Program Metadata

A full understanding of a program can never come from reading only the code. Comprehension requires a knowledge of numerous items of metadata describing the context in which the program was created and is used. Unlike comments, which usually describe a piece of a program, these metadata refer to the entire program. A partial list of program metadata follows:

— Title of program

— Author(s)

— Further developer(s)

— Maintainer(s)

— Owner(s)

— Publisher(s)

— User(s)

— In addition to names for all of the above individuals, their faces, affiliations, postal and network addresses, and phone numbers

— Location of source code, i.e., machine, directory, file(s)

— Version, revision number

— Date and time of this version or revision

— Date and time that the current listing was created

Metadata appear in the program on the title page(s), table(s) of contents, and indices, and in the headers of individual program pages.

Related to but distinct from the metadata are longer texts that describe the program, such as an abstract, statement of purpose, and history. These tertiary texts are described in Sections 3.2, 3.5, and 3.6.

Section 4.2 The Spatial Composition of Comments

Traditional methods of structuring programs pay little attention to developing and enhancing the content and method of presenting comments in relationship to code. Comments, if added at all, are often an afterthought, an unpleasant reminder that management is concerned about issues of program readability and maintainability. Nor is the process of creating comments and integrating them with code facilitated by the interactive text editors and program development environments commonly available.

In our research we were unable to deal with the management issues implied by the legislation of adequate comments nor with the literary and stylistic concerns of making comments both appropriate and meaningful. Instead, we have been concerned with presenting comments for maximum effect, both in isolation and in relationship to code.

To distinguish and highlight comments, we have distinguished external comments (those outside a function definition), internal comments (those within a function definition, which appear on their own line in the input text), and marginalia (those within a function definition, but which do not appear on their own line). The typographic variations that we have considered or explored include:

— Comments integrated with code in a one column format; comments strictly separated from code in a two column format; and various mixtures of one column and two column formats.

— Assuming a two column format, code on the left with comments on the right, or code on the right with comments on the left.

— Assuming a two column format, variations in the width of the code in relation to the width of the comments, for example, 2:1 or 3:1.

— Use of the same font for code and comments, use of variations of one font (roman, bold, italic), and use of three different fonts (for example, a square-serif font such as American Typewriter, a serif font such as Times Roman, and a sans-serif font such as Helvetica).

— Variations in the point size and leading of the comments relative to the point size of the code.

— Use of various diagrammatic notations, such as leader lines,

arrows, or connecting braces, to indicate connectivity between code and comments.

— Use of various gray scale tints overlayed on regions containing various kinds of comments.

— Use of various kinds of rules and boxes to delimit regions containing various kinds of comments.

Section 4.3 The Typography of Punctuation

The punctuation marks of computer programs consist of separators such as ";" and ",", containment symbols such as "(" and "}", and operators such as ".", "!", and "!=". The legibility of punctuation marks in program text is a critical component affecting the comprehensibility of a program, much more so than the legibility of English language punctuation affects the comprehensibility of a passage in English.

We have therefore considered or experimented with various methods of enhancing the legibility of program punctuation, including:

— Emboldening and/or enlarging punctuation marks.

— Kerning compound (multicharacter) operators.

— Substituting symbols that are more legible.

It is obvious that, for C code, the ratio of punctuation marks to alphabetics and numerics is quite different than for prose text. Unfortunately, no typeface currently exists that has been optimized for use in representing computer programs.

Section 4.4 Typographic Encodings of Token Attributes

Current attempts at program visualization often employ crude mechanisms for distinguishing typographically one kind of token from another. Reserved words are often shown in bold face; manifest constants are often named using capital letters only. These attempts, typical of many prettyprinting programs, represent but a small fraction of the wealth of the purely typographic possibilities for enhancing the legibility and readability of programs. The optimum encoding is a complex synthesis of the reader's needs for clarity when scanning the text with a variety of search motives and when examining the text slowly and in detail. Unfortunately, extensive data on programmer's reading patterns is not yet available in the literature of computer science or visible language.

We have experimented with mappings from C token attributes to typographic attributes. We first organized C token attributes according to a token hierarchy. This procedure allowed us to distinguish typographically the following classes:

- Comments (see Section 4.2)
  - External comments
  - Internal comments
  - Marginalia comments
- Punctuation tokens (see Section 4.3)
  - Separator symbols
  - Containment symbols
  - Operators
    - Simple operators
    - Compound operators
- Other tokens
  - Reserved words
    - Preprocessor reserved words (see Section 4.5)
    - Declarative reserved words
    - Control reserved words
    - Control flow altering reserved words
  - Variables
    - Local variables
    - Global variables
    - Static variables
  - Preprocessor macro names
    - Manifest constants
    - Other macros

Other identifiers
- Function names in declarations
- Function names in use
- Typedef names
- Type tags
- Structure and union tags
- Structure and union member names
- Enumeration tags
- Enumeration constants
- Statement labels

Constants
- Integer, floating point, and character constants
- String constants

We then considered or experimented with the visible language appearance of these token attributes to achieve optimum legibility and readability. Attributes used in the encodings included the following:

— Choice of typeface, for example, Helvetica, Times Roman, or American Typewriter.

— Choice of weight, for example, medium or bold.

— Choice of proportion, for example, condensed, normal, or extended.

— Choice of slant, for example, roman or italic.

— Choice of point size, for example, 8, 10, or 14 point.

— Use of capitals or lower case, for example, all capitals, all lower case, initial capitals, small capitals, embedded capitals, and standard prefixes (such as "#").

— An overlayed gray screen tint, or reversed type (white on black).

Section 4.5

The Presentation of Preprocessor Commands

The lexical structure of C encodes all preprocessor commands with a prepended "#". In addition, a standard convention for C programming is the use of all capitalized letters to differentiate preprocessor identifiers (such as manifest constants) from all other tokens.

We have considered or experimented with additional encoding and differentiation, for example:

— Use of typographic attributes such as described in the preceding section.

— Use of positional encodings such as locating all preprocessor commands at the left margin or even exdenting them so that the "#" is in the margin.

— Use of definitional encoding, i.e., showing the macro call in relationship to the text into which it expands.

Section 4.6 The Presentation of Declarations

Thus far we have considered only a program's imperative statements, i.e., statements that transform existing data to produce new data. However, much of a program's intractability often occurs in the declarative aspects, i.e., the declaration of variables as instances of particular data types and the initialization specifying values for certain variables. Again, the issue is complicated by the fact that programs are often scanned for a variety of motives.

We considered or experimented with various methods of using rules and tabular typesetting to enhance the legibility and readability of complex C data declarations, type definitions, and data initialization. These typographic techniques included:

— Consistent use of line spacing, underline rules, and gray screen tints to distinguish sequences of similar lines.

— Multi-column setting of long sequences of short declarations or of lengthy initialization text.

— Tabular setting of sequences of declarations of variables of simple type.

— Tabular setting of declarations of variables of complex type.

Section 4.7

The Visual Parsing of Expressions

One of the most difficult aspects of the detailed reading of a computer program occurs in the attempt to parse a complex (arithmetic or logical) expression. This is particularly true in the programming language C, where 46 different operators occur at 16 levels of precedence, some associating left to right, others associating right to left [Harbison & Steele, 1984]. Current methods of program visualization provide little help to the reader trying to decipher an expression other than the explicit indication of nesting and grouping through the inclusion of parentheses. The resulting visual clutter and masking of what is essential is readily apparent in languages such as LISP.

We considered or experimented with various methods of using typographic attributes to enhance the legibility and readability of complex C expressions. These typographic techniques included:

— Use of ligatures, kerning, and other controls over letter spacing to bind tokens together more tightly.

— Controls over word spacing.

— Variations of the point size of operators.

— Variations of the weight of operators.

— Control over the vertical placement of unary operators.

— Variations in the point size of parentheses.

— Use of light square under-brackets or other diagrammatic notations.

— Explicit introduction of line breaks.

— Control over the vertical placement of phrases.

Section 4.8 The Visual Parsing of Statements

Another vital carrier of the meaning of a program is the syntactic structure of program statements. Statements within a typical C program may nest recursively. At any level, statements such as the *if*, *do...while*, and *switch* contain several component expressions or statements that must be parsed and understood in order that the statement as a whole may be understood. The resulting configuration of separate and nested statements presents a challenge to effective spatial structuring.

We considered or experimented with various methods of applying visible language attributes to enhance a reader's ability to parse complex C statements. These attributes included:

— The amount of indentation used in visually encoding the nesting of phrases within statements, for example, 1, 2 or 3 picas for each level of indentation.

— If there are more than 3 or 4 levels of indentation, clustering of 3 or 4 adjacent levels into groups, distinguishing the groups by larger indentations, rules, leader lines, gray screen tints, or other visual devices. The indentation of a group could be, for example, 8, 10, or 12 picas from the left margin of the preceding group.

— The horizontal position of a left brace, e.g., all the way to the left, hierarchically aligned with the text on the "current line", at the end of the text on the "previous line", and all the way to the right. In the cases of positioning braces in a channel of their own to the left or the right, the braces can be indented within the channel various amounts to encode the hierarchy level.

— The vertical position of the left brace, e.g., the "previous line", between the previous line and the "current line", or the current line.

— The horizontal position of a right brace, e.g., all the way to the left, at the end of the text on the "current line", and all the way to the right. In the cases of positioning braces in a channel of their own to the left or the right, the braces can be indented within the channel various amounts to encode the hierarchy level.

— The vertical position of the right brace, e.g., the "current line", between the current line and the "next line", or the next line.

— Removal of braces altogether, thereby relying upon precise

indentation only to encode visual hierarchy. Alternatively, replacement of braces with a new diagrammatic notation using arrows, pointing symbols, nested brackets, parallel vertical lines, or channels of varying gray value.

— Suppression of line breaks normally introduced where statements are very short.

— Placement of line breaks according to various rules and heuristics, for example, where the line "runs off the edge", before or after an operator of low precedence such as "||" or ",", or such as to create a set of "similar" lines.

— The amount of indentation used after a line break, in various increments finer than the amount of indentation used to encode new levels.

— The amount of line spacing used between segments of a broken line, starting with the standard line spacing and decreasing it slightly by one or two points.

— The use of various diagrammatic notations to indicate continuity with segments of a broken line, such as arrows, ellipses, or regions of gray value.

— The use of various diagrammatic notations such as pointing figures to indicate "unusual" control constructs. A definition of this concept for C might be any label, any *goto* statement, any *continue* statement, any *break* statement not at the end of a *case*, any statement ending a *case* that is not a *break* statement, and any *return* statement not at the end of a function definition.

Section 4.9

The Presentation of Function Definitions

We also had to develop mechanisms to highlight the program's constituent structure in terms of its internally defined functions. The presence of functions help determine for the reader the general sequence and rationale for the program's structure. Making these major "chunks" of the program immediately accessible can contribute significantly to the program's readablility. We considered or experimented with the following techniques:

— Use of pagination to minimize the splitting of function definitions across page boundaries in ways that result in placing most of the text on one page and only a few lines on a subsequent page.

— Use of rules of varying weights under the declaration of the function name and formal parameter list.

— Use of rules of varying weights under the last declaration of a formal parameter.

— Use of headlines for the declaration of the function name and formal parameter list.

— Placement of the type of the value returned by the function, if any, on a line separate from the function name and formal parameter list.

Section 4.10 The Presentation of Program Structure

A C program consists of one or more C source files. Each source file contains a portion of the entire C program, some number of top-level-declarations. These top-level-declarations are either declarations of identifiers used in the program or function definitions elaborating the meaning of new C procedural constructs called functions by defining them in terms of existing C constructs.

SEE, the visual C compiler, produces a listing of a file with respect to a set of included external files binding the external references. These included header files typically contain declarations of identifiers, functions, manifest constants, and new defined types. The declared functions are often defined in "standard libraries" which are stored on the system and which contain functions generally useful to all C programmers.

We considered or experimented with the following techniques:

— Highlighting the global variables by a variety of typographic methods as in Section 4.4.

— The use of a novel mechanism to aid the reading of complex programs structured as a collection of files by adding to each program page footnotes that contain cross-references indicating where in an included file an external identifier is defined and where each identifier defined on a page is used. This produces, in essence, a cross-reference listing distributed throughout the entire program on pages where it is relevant.

Chapter 5 Conclusions

The previous chapters have presented a classification of issues affecting program legibility and readability. We have seen that there are complex interactions of visible language attributes both among themselves and in relation to the C programming language. Despite this, the task of developing a recommended form has proven to be tractable, and we have been able to do many experimental variations before suggesting an optimum appearance.

Based on our work, we believe that a comprehensive, consistent, and effective presentation of a graphic design schema for the appearance of C is desirable to improve program legibility and readability, that we have demonstrated the feasibility of developing such a schema, and that a graphic design manual for the visible language characteristics is an appropriate vehicle in which to present the resulting recommended conventions. As more programmers use the conventions, as they are refined and improved through this use, and as more human factors knowledge about program literature becomes available, the conventions will mature into effective standards.

In achieving this set of objectives, we have also encountered many unforeseen conceptual and technical difficulties. When we began our project, we originally desired a solution for the general problem of typographic and non-typographic representation of programming languages for formats that were both static and those that were dynamic i.e., in an interactive environment. We soon realized that even the more restricted problem of determining static, typographic representations was a challenge. At the time, a wide variety of laser printer fonts of high quality was not readily available, and it was difficult to create even manually composed pages. We have also had to combat a great deal of additional recalcitrant technology (see Chapter 6).

The approach and many of the concrete recommendations for C can be transferred to other languages, such as Pascal and Ada. We must advise those attempting such designs, however, that the task will require extremely careful attention to each language's unique characteristics. By studying these characteristics, it will be possible to design effective visualizations that take advantage of visible language and of the computer language's full potential.

One of the primary difficulties encountered in making graphic design evaluations is that our knowledge of detailed reading motivations and strategies in programmers is limited (see Chapter 6). As a result, it is not yet possible to base decisions among approximately equivalent appearances on any scientific criteria. Nevertheless, we believe that our general methodology is sound, and that our results are significant improvements.

Were we to have merely designed unique prototypes for improvement, this would have had some value. However, we have gone beyond this to provide a tool for generating automatically improved appearance for most C programs. In addition, because it is likely that our conventions will change over the coming years, we have also provided a flexible tool for editing and refining the appearance of these automatically produced program visualizations. Our SEE compiler is one of the most elaborately tunable visible language processing engines available, building as it does both upon the technology of the Portable C Compiler [Johnson, 1979] and upon all of TROFF's text manipulation capabilities. We have pushed these tools as far as they can go in directions for which they were never intended. Future developers will therefore need to provide SEE's functionality (see Volume 6) in a far more appropriate and robust implementation than our prototype.

Thus our approach and our accomplishment have been to design both the best possible appearance for the C programming language within technical and time constraints as well as a suitable prototype of an effective tool for automating, editing, and refining this appearance.

The details of our future research directions are detailed in the next chapter.

Chapter 6 Future Research

Program Visualization Algorithms

There are a number of area fundamental to the enhanced presentation of source text that we have not yet automated. These are the automatic introduction of white space, appropriate automatic line breaking, appropriate automatic page breaking, incorporation of programmer formatting intentions, display of pragmatics, display of diagrammatic representations, and comprehensive automatic warnings and annotations.

Good programmers add blank lines (white space) to enhance the readability of their code. A program visualizer must do this automatically and correctly. An effective algorithm will note the transitions between different kinds of program source text, classifying each line as a comment, a preprocessor command, a component of a function header, a statement within a function body, a component of a type definition, and a component of any other kind of declaration. It will then introduce white space between a line of one kind and a line of another kind. Exactly how much space should be introduced for each kind of transition, as well as the special cases not handled by this simple procedure, must be a subject for future research.

No matter how much space exists for a line on a page, some programmers will write some statements that will need to be "broken" and wrapped to the next line. The result is of course ugly (see Figure 5 of Volume 3), but an appropriate line breaking algorithm.can minimize the visual chaos and damage that results. An effective algorithm will scan backwards from the point representing the most text that will fit on the line, will examine the precedence of the operators that precede that point, and will try to find an operator of "relatively low" precedence that is not "too far" from that point as the place at which to make the break. The algorithm will be complicated by the occurrence of long string constants and will have particular difficulty with lines that begin very deeply indented.

Automatic page breaking and pagination is an even more difficult problem. An implementation problem with the current generation of text formatters (see below) is the need for a great deal of lookahead in order to do the page breaking properly. There are also severe conceptual problems. The basic idea is that there should ideally never be less than three lines in a related "group" of statements

at the top or the bottom of the page. The notion of a group here is related to the concept of the "kind" of source text line defined two paragraphs above. The algorithm becomes difficult because it is not always possible to fulfill this condition, because we want to break the page at a point that is as shallowly nested as possible, because we want to avoid separating an external or internal comment from the code following it to which it typically refers, and because we want at almost any cost to avoid breaking in places such as in the middle of a function header, a *typedef* definition, or a structure definition.

An alternate approach to the optimization of line breaking and page breaking and to the very difficult unsolved problem of the effective display of initializers is the incorporation of programmer formatting intentions. In other words, the visualizer should heed the directions of the programmer when she inserts carriage returns in the middle of statements, extra carriage returns between statements or function definitions, and tabs or carriage returns in the middle of expressions or initializers. How to reconcile these specifications with the default automated decisions of the visualizer is a subject for future research.

Another important topic is the display of pragmatics, features of the code in use. A good example is the need to know what code has changed since the last version. An effective algorithm may employ conventions such as the use of a new font or a gray background to highlight code that has been added, and a diagrammatic convention such as a strike-through line to show where code has been deleted and what has been removed.

We have in our work not yet touched on the possibilities for and the problems in the automatic generation of effective diagrammatic representations. There is a rich variety of techniques to be considered (see, for example, [Martin & McClure, 1985]). Future research is required to select the most valuable representations, and to devise algorithms for automatic conversion between source code and diagram.

Finally, the introduction of fingers pointing at "abnormal" control flow illustrates the need to develop mechanisms for the automatic addition of warnings and annotations. Other examples are the conditions currently detected by the LINT program [Johnson, 1978]. These include unusued variables and functions, variables used before they are set, unreachable parts of the program, and mismatches between function declarations and uses in terms of the

the number and types of arguments. Researchers in *automatic programming* will be able to propose far more substantive ways in which a *programmer's assistant* can detect features of a program and write its suggestions on the listing for consideration by the programmer.

Visualization of other Programming Languages

Our work needs to be extended to programming languages other than C.

The extension to other ALGOL-like languages, e.g., PASCAL and ADA, will be straightforward. The most significant area where some conceptual work may need to be done could be in the effective representation of multi-tasking in ADA.

Languages for artificial intelligence work, e.g., LISP, PROLOG, and SMALLTALK, may present a greater challenge. Designers will have to combat the sea of parentheses presented by LISP and will need to consider the rich data structures and control flow mechanisms either directly present in these languages or available through their many extensions.

Interactive Enhancements of Source Text

Even more interesting is the extension of this work to the interactive display and manipulation of program source text.

One immediate problem that must be faced is the lower resolution (typically, no more than 100 dots per inch) of these devices. This may require modification of many of the techniques that employ a variety of fonts, styles, and sizes and that employ rules and other diagrammatic devices.

On the positive side, interactive program visualization offers a host of new opportunities to incorporate dynamics, animation, color, and sound. We are no longer faced with the difficult problem of establishing "the best" mapping between token types and typographic styles, for the program can be easily re-displayed with different settings. Even more significantly, we can depict through image dynamics and through animation features of the program *in execution*. This is, quite literally, an entire new dimension of program visualization.

Implementation of Program Visualization Processors

As we have intimated above, there are a great many problems remaining to be solved before a system such as SEE can be implemented with ease.

As is explained in more detail in Volume 6, SEE was implemented by making modifications and extensions to the Portable C Compiler. This did not result in an appropriate and robust implementation. Visual compiling is a very different problem from standard compilation, even though it shares common elements such as the need to do lexical analysis and the need to do parsing. Future investigators must therefore develop an appropriate and effective visual compiler technology.

We have also been handcuffed by the lack of an appropriate document formatting technology. The nature of TROFF's processing of text makes formatting that requires look-ahead, such as line breaking and page breaking, very difficult. Standard TROFF, despite the fact that it is supposed to be "device-independent", is very difficult to port to new hardware and to new fonts. It is also impossible to do conversational, interactive document formatting with TROFF; all text must be processed from the very beginning of the document. To build the most effective program visualization aids, we require that research be done on all three of these problems.

As we have indicated, program visualization requires fonts chosen with great care and attention to the fine detail that occurs in computer program source text. The design of fonts that are optimal for the display of computer programs rather than English prose is therefore another task for future research.

Finally, the design and implementation of interactive visualizers will raise an entirely new set of issues that go beyond those encountered in this work.

The Human Factors of Program Reading

There also remains a broad body of concerns and questions that relate to the need to substantiate experimentally that the methods of presentation we propose are effective in making programs more legible, readable, intelligible, memorable, and maintainable.

We must begin with an investigation into how programmers read, a characterization of the cognitive and perceptual processes that comprise the task. An information processing model of program reading

would greatly assist the design of methods of presentation that facilitate the act of reading.

We must then try to measure if our display conventions make programs more legibile, readable, intelligible, memorable, and maintainable, and, if so, by how much are these measures improved?

Finally, we must investigate in what ways our methods of presentation are better. What aspects of our conventions are helpful, which are harmful, and why?

Appendix A

Bibliography

AT&T Bell Laboratories (1985). *The C programmer's handbook.* U.S.A.: Prentice-Hall Inc.

Baecker, R. & Marcus, A. (1983). On enhancing the interface to the source code of computer programs. *Proc. Human Factors in Computing Systems* (SIGCHI '83), Boston, December 1983, 251-255.

Baker, F.T. (1972). Chief programmer team management of production programming, *IBM Systems Journal, 11* (1), 56-73.

Chaparos, A. (1981). *Notes for a federal design manual.* Washington, D.C.: Chaparos Productions.

Dahl, O.-J., Dijkstra, E.W. & Hoare, C.A.R. (1972). *Structured programming.* London: Academic Press.

Eco, U. (1976). *Theory of semiotics.* Bloomington: Indiana University Press.

Gerstner, C. (1978). *Compendium for literates.* Cambridge: MIT Press.

Gilb, T. (1976). *Software metrics.* Studentliteratur, Lund Sweden.

Grogono, P. (1979). On layout, identifiers and semicolons in pascal programs. *SIGPLAN Notices, 14* (4), 35-40.

Gustafson, G.G. (1979). Some practical experiences formatting pascal programs. *SIGPLAN Notices, 14* (9), 42-49.

Gutz, S., Wasserman, A.I. & Spier, M.J. (1981). Personal development systems for the professional programmer. *Computer,* April 1981, 45-53.

Harbison, S.P. & Steele, Jr., G.L. (1984). *C: A reference manual.* Prentice-Hall.

Higgins, D. (1979). *Program design and construction.* Prentice-Hall.

Hueras, J. & Ledgard, H. (1977). An automatic formatting program for pascal. *SIGPLAN Notices, 12* (7), 82-84.

Johnson, S.C. (1978). LINT, a C program checker. *UNIX Programmer's Manual Volume 2.*

Johnson, S.C. (1979). A tour through the Portable C Compiler. *UNIX Programmer's Manual Volume 2.*

Kernighan, B. & Plauger, P.J. (1976). *Software tools.* Addison-Wesley.

Kernighan, B. & Ritchie, D. (1978). *The C programming language.* Prentice-Hall.

Kernighan, B. (1982). A typesetter-independent TROFF. *Bell Laboratories Computing Science Series Technical Report No. 97*, March 1982.

Leinbaugh, D. (1980). Indenting for the compiler. *SIGPLAN Notices, 15* (5), 41-48.

Lions, J. (1977). A commentary on the UNIX operating system. University of New South Wales, Australia.

Marcus, A. & Baecker, R. (1982). On the graphic design of program text. *Proceedings of Graphics Interface 82*, 302-311.

Martin, J. & McClure, C. (1985). *Diagramming techniques for analysts and programmers.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Miara, R.J., Musselman, J.A., Navarro, J.A. & Schneiderman, B. (1983). Program indentation and comprehensibility. *Comm. of the ACM, 26* (11), 861-867.

Nassi, I. & Schneiderman, B. (1973). Flowcharting techniques for structured programming. *SIGPLAN Notices, 8* (8), 12-26.

Oppen, D.D. (1980). Prettyprinting. *ACM Transactions on Programming Languages and Systems, 2* (4), 465-483.

Organick, E. & Thomas, J.W. (1974). Computer-generated semantics displays. *Proc. IFIP Congress,* Applications Volume, 898-902.

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Comm. of the ACM, 15* (12), 1053-1058.

Perlis, A., Sayward, F. & Shaw, M. (Eds). (1981). *Software metrics: An analysis and evaluation.* MIT Press.

Perlman, G. & Erickson, T.D. (1983). Graphical abstractions of technical documents. *Visible Language, XVII* (4), 380-389.

Pike, R. & Presotto, D.L. (1985). Face the nation. *Proceedings of the Summer 1985 Usenix Conference,* Portland, Oregon, June 1985, 81-86.

Rose, G.A. & Welsh, J. (1981). Formatted programming languages. *Software -- Practice and Experience, 11,* 651-669.

Ross, D. (1977). Structured analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering, 3* (1), 16-34.

Rubin, L. (1983). Syntax-directed pretty printing -- A first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering, 9* (2), 119-127.

Ruder, E. (1973). *Typographie.* New York: Hastings House, Visual Communication Books.

Teitelman, W. (1979). A display oriented programmer's assistant. *Int. Jour. Man-Machine Studies, 11,* 157-187.

Wasserman, A.I. (1981). *Tutorial: Software development environments.* Los Alamitos, CA: IEEE Computer Society Press.

Weizenbaum, J. (1966). Eliza — A computer program for the study of natural language communication between man and machine. *Comm. of the ACM, 9* (1), 36-45.

Wirth, N. (1971). Program development by stepwise refinement. *Comm. of the ACM, 14* (4), 221-227.

Wirth, N. (1977). Modula: A language for modular multiprogramming. *Software--Practice and Experience, 7* (1), 3-35.

Yourdon, E. (1979). *Structured walkthroughs.* Englewood Cliffs, NJ: Prentice-Hall.

DISTRIBUTION LIST

addresses	number of copies
Andrew Chruscicki RADC/COEE	25
DOXT RADC/[redacted] GRIFFISS AFB NY 13441	2
RADC/DAP GRIFFISS AFB NY 13441	2
ADMINISTRATOR DEF TECH INF CTR DTIC-DDA CAMERON STA BG 5 ALEXANDRIA VA 22304-6145	2
RADC/RBE (A FEDUCCIA) GRIFFISS AFB NY 13441	1
RADC/RAC GRIFFISS AFB NY 13441-5700	1
RADC/COTD BLDG 3, ROOM 16 GRIFFISS AFB NY 13441-5700	1
AFCSA/SAMI WASHINGTON DC 20330-5425	1
HQ USAF/SITT WASHINGTON DC 20330	1

DIRECTOR DMAHTC ATTN SDSIM 6500 Brookes Lane WASHINGTON DC 20315-0030	1
OASD (C3I) INFORMATION SYSTEMS ROOM 3E187 WASHINGTON DC 20301-3040	2
HQ AFSC/DLAE ANDREWS AFB DC 20334-5000	1
HQ AFSC/XRK ANDREWS AFB MD 20334-5000	1
HQ SAC/NRI (STINFO LIBRARY) OFFUTT AFB NE 68113-5001	1
HQ SAC/SIPT OFFUTT AFB NE 68113-5001	1
HQ ESC/XPZN SAN ANTONIO TX 78243-5000	1
HQ ESC/DOOA SAN ANTONIO TX 78243-5000	1
TAFIG/IIDO ATTN [illegible] ROBERTSON LANGLEY AFB VA 23665-5000	1
HQ TAC/XPS (STINFO) LANGLEY AFB VA 23665-5001	1

HQ TAC/DOYS LANGLEY AFB VA 23665-5001	1
HQ TAC/DRCC LANGLEY AFB VA 23665-5001	1
HQ TAC/DRCT LANGLEY AFB VA 23665-5001	1
HQ TAC/DRCD LANGLEY AFB VA 23665-5001	1
AFSC LIAISON OFFICE LANGLEY RESEARCH CENTER (NASA) LANGLEY AFB VA 23665-5001	1
HQ TAC/DOFR LANGLEY AFB VA 23665	1
HQ TAC/DOFE LANGLEY AFB VA 23665	1
AFWL/SUL ATTN TECHNICAL LIBRARY KIRTLAND AFB NM 87117-6008	1
HQ AFOTEC (OAWD) Attn Capt. Novack) KIRTLAND AFB NM 87117-7001	1
ASD/ENECA WRIGHT-PATTERSON AFB OH 45433	1

ASD/ENEGT (JOSEPH T. BRADFORD) WRIGHT-PATTERSON AFB OH 45433-6503	1
ASD/ENSID ATTN EUGENE WOLANSKI WRIGHT-PATTERSON AFB OH 45433	1
ASD/AXPM WRIGHT-PATTERSON AFB OH 45433	1
ASD/AFALC/AXAE WRIGHT-PATTERSON AFB OH 45433	1
ASD/XRS WRIGHT-PATTERSON AFB OH 45433	1
AFIT/LDEE - TECHNICAL LIBRARY BUILDING 640, AREA B WRIGHT-PATTERSON AFB OH 45433-6583	1
AFWAL/MLPO ATTN DR. G. E. KUHL WRIGHT-PATTERSON AFB OH 45433-6533	1
AFWAL/MLTE WRIGHT-PATTERSON AFB OH 45433	1
AFWAL/FIES/SURVIAC WRIGHT-PATTERSON AFB OH 45433	1
AFAMRL/HE WRIGHT-PATTERSON AFB OH 45433-6573	1

AFHRL/LRS-TDC WRIGHT-PATTERSON AFB OH 45433-6503	1
ASD/EN (CRFP) ATTN MR JEFFERY L PESLER WRIGHT-PATTERSON AFB OH 45433	1
AFHRL/OTS WILLIAMS AFB AZ 85240-6457	1
1843EIG/EIEXM WHEELER AFB HI 96854	1
AUL/LSE 67-342 MAXWELL AFB AL 36112-5564	1
HQ SPACECOM/XPYX ATTN DR WILLIAM R MATOUSH PETERSON AFB CO 80914-5001	1
HQ ATC/TTQI RANDOLPH AFB TX 78148	1
HQ ATC/TTQE RANDOLPH AFB TX 78148	1
CODE H396RL TECHNICAL LIBRARY DEFENSE COMMUNICATIONS ENGINEERING CENTER 1860 WIEHLE AVENUE RESTON VA 22090	1
COMMAND CONTROL AND COMMUNICATIONS DIV DEVELOPMENT CENTER MARINE CORPS DEVELOPMENT & EDUCATION COMMAND ATTN CODE DIOA QUANTICO VA 22134	1

AFLMC/LGY
ATTN CH, SYS ENGR DIV
GUNTER AFS AL 36114 1

COMMANDER
BALLISTIC MISSILE DEFENSE SYSTEMS COMMAND
ATTN DACS-BMZ-AOLIB
PO BOX 1500
HUNTSVILLE AL 35807-3801 1

CHIEF OF NAVAL OPERATIONS
ATTN OP-941F
WASHINGTON DC 20350-2000 1

COMMANDING OFFICER
NAVAL AVIONICS CENTER
LIBRARY - D/765
INDIANAPOLIS IN 46218 1

COMMANDING OFFICER
NAVAL TRAINING EQUIPMENT CENTER
TECHNICAL INFORMATION CENTER
BUILDING 2068
ORLANDO FL 32813-7100 1

COMMANDER
NAVAL OCEAN SYSTEMS CENTER
ATTN TECHNICAL LIBRARY CODE 9642
SAN DIEGO CA 92152-5000 1

US NAVAL WEAPONS CENTER, CODE 343
ATTN TECHNICAL LIBRARY
CHINA LAKE CA 93555 1

SUPERINTENDENT (CODE 1424)
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5100 1

COMMANDING OFFICER
NAVAL RESEARCH LABORATORY
CODE 2627
WASHINGTON DC 20375 1

NAVELEXSYCOM
PDE-110-33
WASHINGTON DC 20363 1

REDSTONE SCIENTIFIC INFORMATION CENTER 2
US ARMY MISSILE COMMAND
REDSTONE SCIENTIFIC INFORMATION CENTER
ATTN DRSMI-RPRD
REDSTONE ARSENAL AL 35898-5241

ADVISORY GROUP ON ELECTRON DEVICES 2
FTS (FEDERAL COMM SYSTEM)
2Ø1 VARICK STREET 11th FLOOR
NEW YORK NY 1ØØ14

LOS ALAMOS SCIENTIFIC LABORATORY 1
ATTN REPORT LIBRARY
MAIL STATION 5ØØØ
LOS ALAMOS NM 87545

AIR FORCE ELEMENT (AFELM) 1
THE RAND CORP
17ØØ MAIN STREET
SANTA MONICA CA 9Ø4Ø6

Commander 1
HQ Fort Huachuca
TECH REF DIV
ATTN BESSIE BRADFORD
Ft Huachuca AZ 85613-6ØØØ

AF TACOM TEST TEAM/TEO 1
Attn LT JAMES GRAVES
FT HUACHUCA AZ 85635

AEDC LIBRARY (TECH REPORTS FILE) 1
MS-1ØØ
ARNOLD AFS TN 37389-9998

U S DEPARTMENT OF TRANSPORTATION LIBRARY 1
FOB-1ØA SECTION, M-493.2, Room 93Ø
8ØØ INDEPENDENCE AVE S.W
WASH DC 2Ø591

1839 EIG/EIET 1
KEESLER AFB MS 39534-6348

HQ AFCC/DAPL 1
BLDG P-4Ø NORTH RM 9
SCOTT AFB IL 62225-6ØØ1

AWS TECHNICAL LIBRARY FL4414 SCOTT AFB IL 62225-5438	1
485 EIG/EIEXR (DMO) GRIFFISS AFB NY 13441-6348	2
HQ ESD/XRX HANSCOM AFB MA Ø1731	1
HQ ESD/XRW HANSCOM AFB MA Ø1731	1
ESD/OCWP HANSCOM AFB MA Ø1731	1
ESD/XRC (AFSC) HANSCOM AFB MA Ø1731	1
ESD/ALSE ATTN MR. WILLIAM J. LETENDRE BLDG 17Ø4, RM 2Ø6 HANSCOM AFB MA Ø1731	1
ESD/ALSE ATTN CAPTAIN ART DESCLLES BLDG 17Ø4, RM 2Ø6 HANSCOM AFB MA Ø1731	1
ESD/ALSE ATTN CAPTAIN JOE ITZ BLDG 17Ø4 RM 2Ø6 HANSCOM AFB MA Ø1731	1

ESD/ALSC
ATTN MAJOR GEORGE JACKELEN
BLDG 1704 RM 107
HANSCOM AFB MA 01731 1

ESD/TCS-2B
ATTN MAJOR JOSEPH H. SCHMOLL
HANSCOM AFB MA 01731 1

ESD/TCS-1D
ATTN CAPTAIN J. MEYER
HANSCOM AFB MA 01731 1

DIRECTOR
NSA/CSS
ATTN T5112 /TDL (MARJORIE E. MILLER
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN W161
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN R24
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN R21
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN R31
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN R5
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN R6
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN R8
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN S031
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN S21
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN V307
FORT GEORGE G MEADE MD 20755-6000 1

Aaron Marcus and Associates
1196 Euclid Avenue
Berkeley CA 5

DoD COMPUTER SECURITY CENTER
ATTN C42, TIC
9800 SAVAGE ROAD
FORT GEORGE G MEADE MD 20755-6000 1

DIRECTOR
NSA/CSS
ATTN L-222
FORT GEORGE G MEADE MD 20755-6000 [illegible]

Dr Ron Baecker
Human Computing Resources Corp
10 St Mary Street
Toronto, Ontario
Canada M4Y 1P9 5

TOTAL COPIES REQUIRED 134

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.

THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.